2’s Complement Properties

• As with sign and magnitude, leading 0s ⇒ positive, leading 1s ⇒ negative
  - 000000...xxx is ≥ 0, 111111...xxx is < 0
  - except 1...1111 is -1, not 0 (as in sign & mag.)

• Only 1 Zero!

2’s Complement Number “line”: \( N = 5 \)

- \( 2^{N-1} \) non-negatives
- \( 2^{N-1} \) negatives
- one zero
- how many positives?

Two’s Complement Formula

• Can represent positive and negative numbers in terms of the bit value times a power of 2:
  \[ d_{31} \times (-2^{31}) + d_{30} \times 2^{30} + \ldots + d_1 \times 2^1 + d_0 \times 2^0 \]

• Example: 1101\text{two}
  \[ = 1 \times (-2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \]
  \[ = -8 + 4 + 0 + 1 \]
  \[ = -3_{\text{ten}} \]

Two’s Complement shortcut: Negation

• Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result

Proof*: Sum of number and its (one’s) complement must be 111...111\text{two}

However, 111...111\text{two} = -1\text{ten}

Let \( x' \) ⇒ one’s complement representation of \( x \)

Then \( x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x \)

• Example: -3 to +3 to -3\text{x}: 1111 1111 1111 1100\text{two}
  1111 1111 1111 1111 1111 1111 1111 1100\text{two}

* Check out www.cs.berkeley.edu/~dsw/twos_complement.html

Two’s comp. shortcut: Sign extension

• Convert 2’s complement number rep. using \( n \) bits to more than \( n \) bits

  Simply replicate the most significant bit (sign bit) of smaller to fill new bits
  - 2’s comp. positive number has infinite 0s
  - 2’s comp. negative number has infinite 1s
  - Binary representation hides leading bits; sign extension restores some of them

  -16-bit -4\text{ten}, to 32-bit:

  \[
  \begin{array}{cccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  \end{array}
  \]

0000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000
What if too big?

- Binary bit patterns above are simply representatives of numbers. Strictly speaking they are called "numerals".
- Numbers really have an infinity number of digits
  - with almost all being same (00...0 or 11...1) except for a few of the rightmost digits
  - Just don’t normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, overflow is said to have occurred.

```
00000 00001 00010 11110 11111
```

Number Summary

- We represent “things” in computers as particular bit patterns: \( N \text{ bits } \Rightarrow 2^N \)
- Decimal for human calculations, binary for computers, hex to write binary more easily
- 1’s complement - mostly abandoned

```
00000 00001 00010 11110 11111
```
- 2’s complement universal in computing: cannot avoid, so learn

```
00000 00001 00010 11110 11111
```

Preview: Signed vs. Unsigned Variables

- Java just declares integers `int`
  - Uses two’s complement
- C has declaration `int` also
  - Declares variable as a signed integer
  - Uses two’s complement
- Also, C declaration `unsigned int`
  - Declares an unsigned integer
  - Treats 32-bit number as unsigned integer, so most significant bit is part of the number, not a sign bit

BIG IDEA: Bits can represent anything!!

- **REMEMBER**: \( N \) digits in base \( B \) \( \Rightarrow B^N \) values
  - For binary in particular: \( N \text{ bits } \Rightarrow 2^N \) values
- Characters?
  - 26 letters \( \Rightarrow 5 \text{ bits } (2^5 = 32) \)
  - upper/lower case + punctuation \( \Rightarrow 7 \text{ bits in 8} \) (“ASCII”)
  - standard code to cover all the world languages \( \Rightarrow 16 \text{ bits } (“Unicode”) \)
- Logical values?
  - 0 \( \Rightarrow \) False, 1 \( \Rightarrow \) True
- Colors? Ex: Red (00) Green (01) Blue (11)
- Locations / addresses? commands?

Example: Numbers represented in memory

- Memory is a place to store bits
  - A word is a fixed number of bits (eg, 32) at an address
- Addresses are naturally represented as unsigned numbers in C

Disclaimer

- **Important**: You will not learn how to fully code in C in these lectures!
  You’ll still need your C reference for this course.
  - K&R is a great reference.
    - But… check online for more sources.
  - “JAVA in a Nutshell,” O’Reilly.
    - Chapter 2, “How Java Differs from C”.
Compilation: Overview

C compilers take C and convert it into an architecture specific machine code (string of 1s and 0s).
- Unlike Java which converts to architecture independent bytecode.
- Unlike most Scheme environments which interpret the code.
- Generally a 2 part process of compiling .c files to .o files, then linking the .o files into executables.

Compilation: Advantages

- Great run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture).
- OK compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled.

Compilation: Disadvantages

- All compiled files (including the executable) are architecture specific, depending on both the CPU type and the operating system.
- Executable must be rebuilt on each new system.
  - Called "porting your code" to a new architecture.
- The "change → compile → run [repeat]" iteration cycle is slow.

C vs. Java™ Overview (1/2)

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-oriented (OOP)</td>
<td>No built-in object abstraction. Data separate from methods.</td>
</tr>
<tr>
<td>&quot;Methods&quot;</td>
<td>&quot;Functions&quot;</td>
</tr>
<tr>
<td>Class libraries of data structures</td>
<td>C libraries are lower-level</td>
</tr>
<tr>
<td>Automatic memory management</td>
<td>Manual memory management</td>
</tr>
<tr>
<td></td>
<td>• Pointers</td>
</tr>
</tbody>
</table>

C vs. Java™ Overview (2/2)

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>High memory overhead from class libraries</td>
<td>Low memory overhead</td>
</tr>
<tr>
<td>Relatively Slow</td>
<td>Relatively Fast</td>
</tr>
<tr>
<td>Arrays initialize to zero</td>
<td>Arrays initialize to garbage</td>
</tr>
<tr>
<td>Syntax: /* comment */ // comment System.out.print</td>
<td>Syntax: /* comment */ printf</td>
</tr>
</tbody>
</table>

C Syntax: Variable Declarations

- Very similar to Java, but with a few minor but important differences.
- All variable declarations must go before they are used (at the beginning of the block).
- A variable may be initialized in its declaration.
- Examples of declarations:
  - correct: 
    ```c
    int a = 0, b = 10;
    ...
    ```
  - incorrect: 
    ```c
    for (int i = 0; i < 10; i++)
    ```
C Syntax: True or False?

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (pointer: more on this later)
  - no such thing as a Boolean
- What evaluates to TRUE in C?
  - everything else...
  - (same idea as in scheme: only #f is false, everything else is true!)

C syntax : flow control

- Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
  - if-else
  - switch
  - while and for
  - do-while

C Syntax: main

- To get the main function to accept arguments, use this:
  ```c
  int main (int argc, char *argv[])
  ``
- What does this mean?
  - argc will contain the number of strings on the command line (the executable counts as one, plus one for each argument).
    - Example: `unix% sort myFile`
  - argv is a pointer to an array containing the arguments as strings (more on pointers later).

Administrivia

- First labs today (“lab is where the learning happens”)
- Office hours are still being arranged
- Class Newsgroup
  - ucb.class.cs61c

Address vs. Value

- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
- Don’t confuse the address referring to a memory location with the value stored in that location.

Pointers

- An address refers to a particular memory location. In other words, it points to a memory location.
- Pointer: A variable that contains the address of another variable.
Pointers

• How to create a pointer:
  & operator: get address of a variable
  
  int *p, x;
  p = &x;

• How to change a variable pointed to?
  * use dereference * operator on left of =
  
  int x = 3;
  *p = 5;

• How get a value pointed to?
  * dereference operator: get value pointed to
  
  printf("p points to %d\n",*p);

Pointers and Parameter Passing

• Java and C pass a parameter “by value”
  • procedure/function gets a copy of the parameter, so changing the copy cannot change the original
  
  void addOne (int x) {
    x = x + 1;
  }
  int y = 3;
  addOne(y);

  • y is still = 3

Pointers

• Normally a pointer can only point to one type (int, char, a struct, etc.).
  • void * is a type that can point to anything (generic pointer)
  • Use sparingly to help avoid program bugs... and security issues... and a lot of other bad things!

And in conclusion...

• All declarations go at the beginning of each function.
  • Only 0 and NULL evaluate to FALSE.
  • All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
  • A pointer is a C version of the address.
    • * “follows” a pointer to its value
    • & gets the address of a value